







Service Function Chaining Design & Implementation Using Network Service Mesh in Kubernetes

Abdullah Bittar¹^(✉), Ziqiang Wang¹, Amir Aghasharif¹,
Changcheng Huang¹, Gauravdeep Shami², Marc Lyonnnais²,
and Rodney Wilson²

¹ Carleton University, Ottawa, ON K1S 5B6, Canada
{abdullahbittar, ziqiangwang, amiraghasharif, huang}@carleton.ca

² Ciena Corporation, Ottawa, ON K2K 0L1, Canada
{gshami, mlyonnai, rwilson}@ciena.com

Abstract. Service Function Chaining (SFC) in a cloud-native environment is becoming essential as more users move towards clouds today. Cloud-native environments utilize container-based microservices to provide software solutions. Integrating SFC with container-based microservices introduces new challenges. This paper exploited Network Service Mesh (NSM) framework features to create a service function chain on a multi-node Kubernetes cluster. We focus on the design and implementation of SFC in Kubernetes using NSM. Also, we deployed our custom-built containers in the Kubernetes cluster to create a service function chain. Hence, we demonstrate how an SFC is designed in a cloud-native environment rather than a traditional NFV/SDN approach. Furthermore, to evaluate the performance, we compare different frameworks that support SFC in Kubernetes, highlighting the advantage and disadvantages of each framework.

Keywords: Service Function Chain · SFC · Kubernetes · Network Service Mesh · NSM · Design · Implementation

1 Introduction

Next-generation networks mainly rely on the virtualization of network functions [1]. The network virtualization concept affects network operations, deployment, and expansion, especially by leveraging its benefits. Network Function Virtualization (NFV) provides some benefits such as scalability, flexibility, and cost. The idea behind NFV is to integrate different network equipment into industry-standard high-capacity servers, storage, and switches that can be located in various locations, such as data centers [2].

NFV introduced new opportunities to exploit the network and provide better services for its users. According to the International Data Group (IDG)

2020 survey, 92% of the organization’s IT environments are at least somewhat in the cloud today. More than half of the organizations currently use multiple clouds [20]. This considerable shift towards cloud-based operation introduces new techniques such as adopting microservice software to bring more flexibility and agility to NFV architecture. This phenomenon brought the so-called Cloud-native Network Function (CNF) [21]. CNF re-designed network functions to become self-contained, transforming them into a container format. This inspiration introduces challenges for supporting new applications, such as identifying and steering traffic for different users or content. Furthermore, cloud-native environments such as Kubernetes do not support NFV networking requirements, such as network isolation and fixed containers IP [31]. Lacking the support of network requirements endangers an essential task in NFV, Service Function Chaining (SFC). SFC is a mechanism that allows multiple different service functions to be connected to form a chain enabling carriers to benefit from the virtualized software-defined infrastructure. Hence, SFC is essential to spawning on-demand network services. In this paper, we concentrate on the design and implementation of SFC in a cloud-native environment, an approach that has not been addressed yet. While traditional SFC solutions have been widely addressed using the Software-Defined Networking (SDN) approach, SFC cloud-native is still somewhat novel in the research community. Our primary goal in this paper is detailing the design and the implementation of SFC concept deployed in a Kubernetes-based solution by leveraging Network Service Mesh (NSM) framework. The contribution of this paper is threefold:

1. We investigate various tools that can be utilized to build an SFC over Kubernetes
2. Inspired by the Network Service Mesh (NSM) approach, we design and implement a multi-node cloud-native SFC framework using a custom-build container in a Kubernetes cluster.
3. Performance analysis and evaluation of various Kubernetes SFC tools

The rest of the paper is organized as follows. Section 2 provides background on key concepts such as NFV, SFC, Kubernetes, containers, Container Network Interface (CNI), NSM, and the motivation behind our work. Section 3 describes our cloud-native traffic steering design. Section 4 provides details about the implementation process. In Sect. 5 we provide performance evaluation, while in Sect. 6, we provide the limitations and future work. Section 7 summarizes the main related work dealing with SFC in Kubernetes, and finally, the last section will conclude the paper.

2 Background

The main idea of NFV is to integrate proprietary network devices into industry-standard high-capacity servers, storage, and switches in various locations such as data centers [2]. The NFV architecture inherently promises advantages such as software and hardware decoupling, flexible network function deployment, and

dynamic functioning [3]. The Internet Engineering Task Force (IETF) describes Service Function Chaining (SFC) as an ordered set of service functions in which they must be executed [4]. Therefore, an SFC would consist of physical or virtualized network functions chained together where traffic will pass through before reaching the destination.

2.1 Motivation

SFC plays a vital role in next-generation networks by benefiting different technologies such as 5G, IoT, and edge computing [9, 10, 39, 40]. SFC helps in providing customizable network function services to traffic flows between different networks. The demand for new network services has grown exponentially, aided by the explosion of new network technologies and infrastructures, such as the success of cloud networks, that have increased the degree of pervasiveness and connectivity between heterogeneous devices. Our motivation behind this paper is to provide the reader with sufficient details regarding the design and implementation of SFC in Kubernetes using the NSM framework while delivering a real-life use case. This approach has not been addressed before in a cloud-native environment tool like Kubernetes. Focusing on chaining containers rather than traditional virtual machines, we created a cluster in Kubernetes that consisted of multiple Pods chained together and added video reduction and video broadcast service functions.

2.2 Kubernetes

Kubernetes is a cloud-based platform that offers a Container-as-a-Service (CaaS) layer for managing containerized workloads and services. According to Kubernetes' main web page, Kubernetes is "an open-source system for automating deployment, scaling, and management of containerized applications" [5]. The simplest way to describe Kubernetes' primary function is container cluster orchestration. Google originally designed Kubernetes, but since 2014 Cloud Native Computing Foundation (CNCF) has been responsible for maintaining Kubernetes. Kubernetes is a decentralized architecture based on a declarative model that defines the ultimate state. Users only describe the application's structure to be deployed when using the declarative model. In contrast, in the imperative model, the user must clearly define all the technical deployment tasks to be performed in sequence order. Kubernetes implement a microservice architecture that considers a single application as a collection of small services, each running in its process and communicating with each other with a lightweight mechanism, typically an HTTP service/gRPC service and corresponding API.

A cluster in Kubernetes consists of master and worker nodes. A node in Kubernetes may be either a virtual or a physical machine. Figure 1 is an example of a Kubernetes cluster that consists of two master nodes and three worker nodes. The main elements in a master node are different than those in a worker node. Intuitively from its name, the worker node performs the actual workload. Applications will have to be containerized, and a Pod will encapsulate the

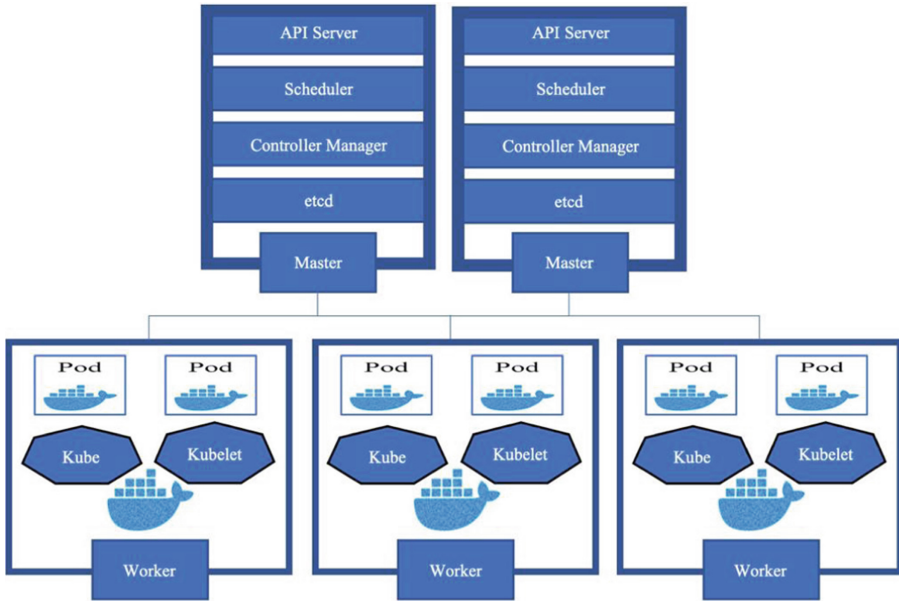


Fig. 1. Kubernetes cluster architecture and the main components.

container. The worker node will have at least one Pod. A Pod is the smallest deployable unit of computing that can be created and managed in Kubernetes. A Pod is a group of one or more containers in which they share resources such as shared storage, networking and information about how to run each container. Pods are temporary but can be automatically re-created to meet the desired state. There are three main background services running in each worker node. The first process is the container runtime which is responsible for running containers. Secondly, the Kubelet agent is the frontend CLI used to communicate with kube-apiserver, which resides on the master node. Thirdly, the Kube Proxy is a network proxy to forward the requests between different elements in the cluster.

A master node is considered a control plane that includes four main processes. The first process is API Server which acts as a cluster gateway and gatekeeper for authentication. All requests initiated by users will first pass through API Server for validation. The second process is Scheduler which is responsible for scheduling Pods on nodes while considering the resources in the cluster. The third process is the Controller Manager which is responsible for detecting cluster changes and bringing the cluster status to its desired state. The last process is etcd which is a key-value store for the cluster state [5].

Kubernetes offers minimal networking services. Kubernetes only provides the networking model placeholder and does not provide networking services/extensions in the cluster. In most cases, Kubernetes depend on third-party projects that provide network functionality. Four different types of network-

ing communication should be addressed when developing a network extension. Firstly, highly-coupled container-to-container communications. Secondly, Pod-to-Pod communications. Thirdly, Pod-to-Service communications and finally is the External-to-Service communications. A Service in Kubernetes is an abstract way to expose an application running on a set of Pods as a network service. Since Pods are ephemeral, Pods are associated with Services through key-value pair, where the Service will automatically discover new Pods with labels that match the key-value pair. Furthermore, Kubernetes imposes fundamental requirements on any networking implementation to allow Pods on a node to communicate with all Pods on all nodes without Network Address Translation (NAT). Third-party projects develop networking extensions that meet the networking module requirements, while each may have a different focus. Hence, users will have to choose amongst the available networking extensions that meet their needs to deploy them in Kubernetes.

2.3 Containers

Self-contained network functions are moved into a container, such as routers or firewalls. With containers, users can pack up their services neatly, including all application binaries, software dependencies, and necessary configuration files. This also means that the application will remain constant regardless of where they are running. Containers incur significantly lower overhead than traditional Virtual Machines (VMs). It is essential to mention that not all virtual network functions are feasible to be containerized [7]. Containers help businesses modernize by making it easier to scale and deploy applications. According to a CNCF survey done in 2020, containers usage in production has increased 300% since 2016, and 92% of users surveyed say they use containers [6]. Lightweight virtualization technologies such as cloud-native containers are the trend in deploying applications in the cloud infrastructure. Container-native Network Function (CNF) is a software implementation of a network function built and deployed in a cloud-native method [8]. Despite all the benefits gained from integrating containers into the NFV environment, there will be management and orchestration challenges that may hinder the utilization of container-based VNFs. Containers introduce new challenges and complexity by introducing an entirely new infrastructure ecosystem.

2.4 Different CNI Plug-ins

Networking in Kubernetes is provided by the so-called Container Network Interface (CNI), a CNCF project that defines the configuration of network interfaces for Linux containers. CNI comprises specifications and libraries for plug-ins to configure network interfaces in Linux containers [11]. A unique file called the CNI plug-in is responsible for inserting the correct network interface into the container network while making any necessary changes on the host. There are different kinds of CNIs, and each one provides a particular behaviour to allow networking inside the cluster. Some of the most common CNIs are Calico and

Canal, according to [12]. Firstly, Calico is well known for its performance, flexibility, and power. Calico provides additional functions, such as network security and administration, and essential Pod to Pod connections [13]. Secondly, Canal integrates Calico and another CNI called Flannel into one CNI to deploy in a Kubernetes cluster. It uses Flannel for networking pod traffic between hosts via VXLAN and Calico for network policy enforcement and Pod to Pod traffic [14]. Weave Net is another CNI plug-in. It is resilient and straightforward to use the network for Kubernetes and its hosted applications [42]. Weave Net creates a virtual network that connects Docker containers across multiple hosts and enables their automatic discovery. One of Weave Net's benefits is that it comes with a Network Policy Controller that automatically monitors Kubernetes for any NetworkPolicy annotations on all namespaces and configures iptables rules to allow or block traffic as directed by the policy

2.5 Network Service Mesh (NSM)

Kubernetes' principle includes service discovery and load balancing in an automated function for scaling up or down applications. On this basis, Kubernetes does not focus on the networking aspect but on managing a cluster. Kubernetes cannot provide some advanced L2/L3 network features, and it lacks the support for cross-cluster connectivity. Network Service Mesh (NSM) utilizes Kubernetes' networking model to perform specific networking functions. NSM is a novel approach to solving complicated L2/L3 use cases in Kubernetes that are tricky to solve [17], such as SFC use case. NSM is inspired by Software Defined Networking (SDN), in which NSM maintains the separation between control and data plane while providing network intelligence between microservices.

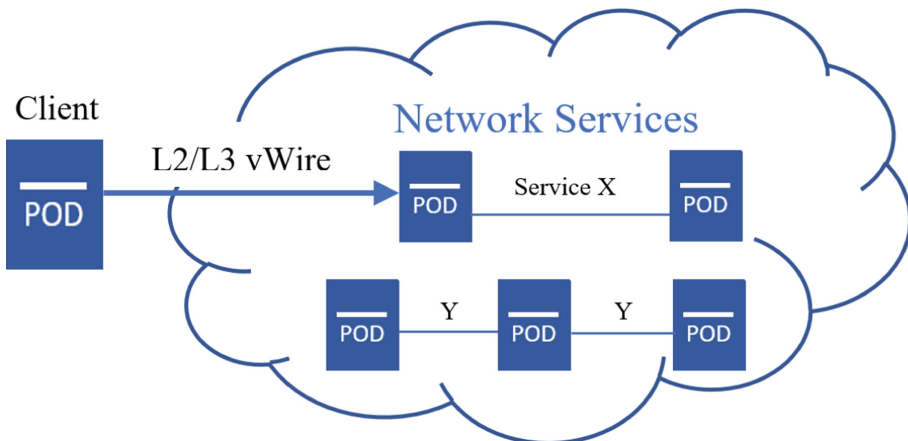


Fig. 2. NSM

NSM is based on three basic concepts. The first concept is Network Service (NS) which provides L2/L3 service. The second concept is Network Service Endpoint (NSE), a Pod in a Kubernetes cluster that provides the NS application. The final concept is the L2/L3 connection between the client's Pod and the NSE(s). NSM extends beyond kernel interface to support complex use cases and provides other interfaces such as memif or vhost-user interfaces. A memif interface, called Shared Memory Packet Interfaces, provides high-performance packet transmit and receive between the user application and Vector Packet Processing (VPP) [41]. NSM allows individuals to connect to an NS independently of the infrastructure they are running on. An NS, such as a chain of microservices, must be identified in a cluster to allow users to access it. After creating an NS, users will request to join a specific NS in the cluster by assigning a Pod to the user and creating a vWire to connect to the NS. User's Pod will have a unique annotation key-value pair that will specify which NS to connect. The NSM Manager will create a vWire that connects the user's Pod to the specified NS. A simple example of NSM connectivity is in Fig. 2. This figure illustrates how a client can access an NS on the Kubernetes cluster. Each NSE (Pod) includes a key-value pair to identify which NS it belongs to. In Fig. 2, a client would like to connect to Service X by sending a request to the NSM Manager. In return, the NSM Manager will examine the annotation key-value pair in the client's Pod and then check if the required NSEs and interface mechanism are available in the cluster. If there is a match, the NSM manager will respond to the request by creating a vWire to the appropriate requested NS.

NSM consists of a few elements that are important for its functionality. Figure 3 provides a graphical representation of the NSM control and data plane elements.

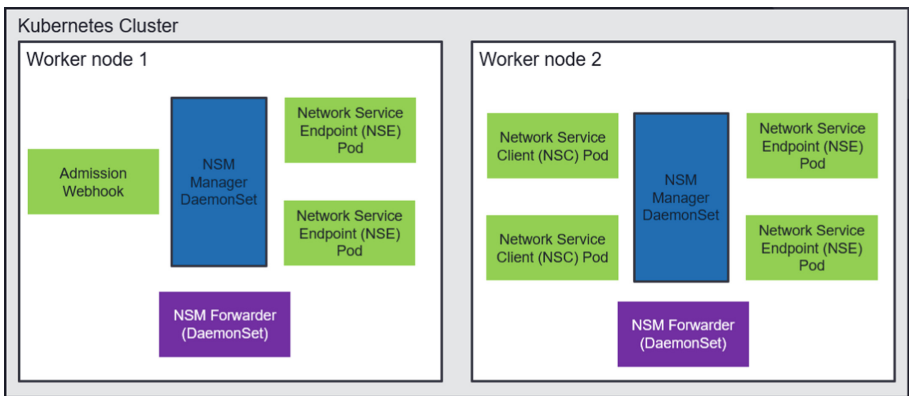


Fig. 3. NSM

Network Service Client is the first element involved in NSM. It is deployed as a Pod in a Kubernetes environment, and its main aim is to require a cross-

connection to a specific NS. On the other hand, NSE oversees implementing network functions in a network service. Both Network Service Client and NSE can be composed of two containers, one container for implementing NSM control plane functionalities and the other container implementing the primary service function. NSM Manager Pod is fundamental in the control plane that consists of three essential containers. The first container, which is the heart of the NSM control plane implementation, is the nsmd container. This container is responsible for all requests that involve cross-connections construction. The second container is nsmd-k8s, responsible for registries between different NSM Managers. The final container is nsmdp container which is in charge of checking that all elements involved in NSM Manager functions are working correctly. Network Service Forwarder's main aim is to implement NSM data-plane functionalities. When communication between Pods is provided, it is in charge of configuring interfaces and building cross-connection between involved Pods. Lastly, Admission Webhook intercepts Pod creation request to api-server and based on its internal configuration, and it can modify the request and inject specific code in the YAML request.

3 Design

In this paper, we address the problem of SFC in microservices-based architecture. Our contribution aims to provide details of the design and implementation process in deploying SFC in the Kubernetes cluster using custom-build containers by leveraging the NSM plug-in while adding extra features. Deploying an SFC in Kubernetes includes three steps: online search for third-party networking extension supporting SFC, deploy correct configurations for creating SFC and building an SFC in Kubernetes.

First Step is to search online for third-party projects (network extension) that support SFC in Kubernetes. Luckily, there are few options available. The first network extension is Contiv-VPP [15] which uses FD.io VPP to provide network connectivity between Pods in a Kubernetes cluster. The FD.io [16] is the world's secure networking data plane project that focuses on supporting terabit software data plane by using the VPP concept, which processes multiple packets at a time with low latency. Contiv-VPP is a CNI plug-in that employs a programmable CNF vSwitch offering SFC and other high-performance cloud-native networking and services. The second network extension is called OVN4NFV-K8s [19], and it is based on an Open Virtual Network (OVN) CNI controller to provide cloud-native-based SFC and other overlay networking features. OVN4NFV-K8s is a project under the Open Platform for NFV (OPNFV), a collaborative open-source platform for NFV. The third and final network extension that supports SFC in Kubernetes is NSM.

We tested the abovementioned three networking extensions to deploy SFC. We were able to deploy an SFC in a Kubernetes cluster using the Contiv-VPP extension successfully. Contiv-VPP provides three different scenarios to deploy

an SFC [45]. The first scenario is adding a tap interface to Linux CNFs. Secondly, each CNF Pod runs its own VPP instance and is connected with one or two additional memif interfaces. The final scenario is connecting a CNF to external Data Plane Development Kit (DPDK) sub-interfaces via two additional memif interfaces. The additional tap/memif interfaces between Pods/external interfaces are inter-connected on the L2 layer, using an L2 cross-connect on the vSwitch VPP. Contiv-VPP may be used on bare metal servers or using VMs. We went with VMs, where Kubernetes and service functions were on different VMs deployed on a single server.

We faced challenges in deploying SFC in OVN4NVF. The main issue was because the coreDNS Service Pod does not initiate. In other words, the API server could not get the endpoint of kube-dns Service. We ensured that no firewall was stopping the traffic and that coreDNS and API configurations were correct and functioning. OVN4NVF provides instructions on how to set up Kubernetes using VMs.

The third extension we tested was NSM, which is entirely orthogonal to standard Kubernetes networking. NSM allows Pods network with different workloads across the cluster using a simple set of APIs designed to provide connectivity, security, and observability. NSM leverages the Custom Resource Definition (CRD) service Kubernetes provides to define a custom resource in a cluster that performs a specific function [18]. NSM introduces an NS CRD, representing the logical implementation of a chain of network functions implemented as Pods in the cluster. The NS also specifies the order of the network function chain in which traffic should follow when traversing. It is also important to mention that the NSM control plane implements a cross-connection between Pods to allow proper communication. The cross-connection comprises two interfaces injected in the Pods involved in the communication.

Second Step is divided into two phases where the first phase is to deploy a cluster in Kubernetes, and the second phase is to configure the cluster according to the networking extension you choose in step one. Deploying a cluster in Kubernetes can be done using different tools. Kubeadm is a tool to build a Kubernetes cluster on a bare metal server [38]. Kubeadm toolbox will bootstrap a minimum viable Kubernetes cluster that conforms to best practice, allowing adding many nodes to the cluster. Another tool to build a cluster is using Kind tool. Kind is an open-source tool that generates Kubernetes clusters using Docker [25]. Kind was primarily designed for testing Kubernetes itself. Kind makes it easy to create a cluster by simply passing the command ‘kind create.’ NSM (release v0.2.0) uses the Kind tool to create clusters by default. Hence, Kind uses Kindnet [26] as the default networking plug-in. Kindnet implements the Kubernetes networking model using the CNI reference plug-ins and uses Docker’s default bridge networking. We created a cluster in Kubernetes that consisted of multiple Pods and services.

The second phase configures the Kubernetes cluster according to the network extension deployed to build an SFC. In general, all networking extensions follow

the same concept to identify an SFC service. Differences are mainly founded in the attribute values of the configuration files. The central idea is to create a CDR and reference the CRD in the services deployed in the cluster. The CRD is used to define an SFC with a name and schema. Figure 4a is a YAML configuration file of our custom CRD based on the NSM framework schema. It identifies a new custom resource that defines the concept of a networking service chain with the name of NetworkServiceChain, which will be used to create a network service chain, as Fig. 4b shows. Figure 4b is a YAML configuration file that identifies a chain of network services. We used the NetworkServiceChain name as an identifier and added it to the ‘kind’ attribute.

Traffic must follow the two matching rules, as depicted in Fig. 4b. Specifically, the first matching rule requires that the forwarder direct all traffic flow from any client that connects with this NS to the ‘firewall’ Pod, the entry point to the chain in the cluster. The matching rule is indicated in the red box in Fig. 4b. The second matching rule requires that traffic flow from the firewall Pod be steered to a second Pod, the ‘vid-reduction’ Pod, as indicated in the orange box in Fig. 4b. Hence, a flow request coming from an NSC Pod will be first headed to the ‘firewall’ Pod then to the ‘vid-reduction’ Pod. The metadata name attribute, ‘SFC-1’, in Fig. 4b is an essential attribute. This is the only method for attaching a Pod to a chain by having the metadata name of the chain in the Pod’s deployment configuration file.

Third Step in building an SFC in Kubernetes is to deploy the Pods in the cluster. This step is container development and adding them to a Pod. Developers need to create containers that will perform their application’s service/network function. After that, containers will be wrapped by Pods in the Kubernetes cluster. Our chain consists of three Pods in a sequence plus an extra Pod for the client, as illustrated in Fig. 5. Inside each Pod, we added a container that we custom-built to perform a specific function:

1. Firewall Pod: a firewall container to detect IP addresses and port numbers.
2. Vid-Reduction Pod: a container that performs video reduction size function.
3. Vid-Broadcast Pod: a container that broadcasts the video to users.

Docker is an open-source platform for building, deploying and managing containerized applications [22]. We used Docker to develop our containers and specifically included a CNI responsible for allocating network interfaces to the newly created container network namespace and making necessary changes on the host to enable the connectivity with other containers on the same network. Using the specification provided by the CNI GitHub [23], an IP address should be assigned to the interface using the correct IP address management.

Network Automation. To complete the design process, it is a good idea to automate the deployment process. There are multiple methods for automation, and one of the methods is to develop coding scripts ready for deployment. In our experiment, we created numerous Python scripts that configure, manage,

```

apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: networkservices.networkservicemesh.io
spec:
  conversion:
    strategy: None
  group: networkservicemesh.io
  names:
    kind: NetworkServiceChain
    listKind: NetworkServiceChainList
    plural: networkservicechains
    shortNames:
      - netsvcch
      - netsvcchs
    singular: networkservicechain
    scope: Namespaced

```

(a) CRD

```

apiVersion: networkservicemesh.io/v1
kind: NetworkServiceChain
metadata:
  name: SFC-1
  namespace: nsm-system
spec:
  payload: ETHERNET
  matches:
    - source_selector:
        app: firewall
      routes:
        - destination_selector:
            app: vid-reduction
    - routes:
        - destination_selector:
            app: firewall

```

(b) Network Service Chain

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: firewall
  annotations:
    ns.networkservicemesh.io: SFC-1
spec:
  template:
    spec:
      containers:

```

(c) Annotation Method

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: firewall
  annotations:
    ns.networkservicemesh.io: NetworkServiceChain
spec:
  template:
    spec:
      containers:
        - name: nsc-firewall
          env:
            - name: NSM_NETWORK_SERVICE
              value: kernel:///SFC-1/nsm-1
            - name: NSM_REQUEST_TIMEOUT
              value: 75s
      nodeSelector:
        kubernetes.io/hostname: ubuntu

```

(d) Environment Variables Method

Fig. 4. Kubernetes configuration files, YAML

and deploy the SFC in the Kubernetes cluster. We also built a simple web page, User Interface (UI), for clients to interact with the cluster to choose a video file from the available list for broadcasting. A request would be sent from the web page (frontend) to the backend to broadcast the requested video. The web page and the backend were developed using Python because of their simplicity in integrating the frontend to the backend process.

The first action the backend performs is creating a Pod for the client. Pod creation is crucial because the newly created Pod will contain metadata to identify which service function chain they would like to connect. There are two different methods to define the service function chain name in the NSM framework. Users can use annotation or include a variable in the environment specifications. The client's Pod configuration file will consist of an attribute called annotation, which specifies the name of the NS or the service function chain they would like

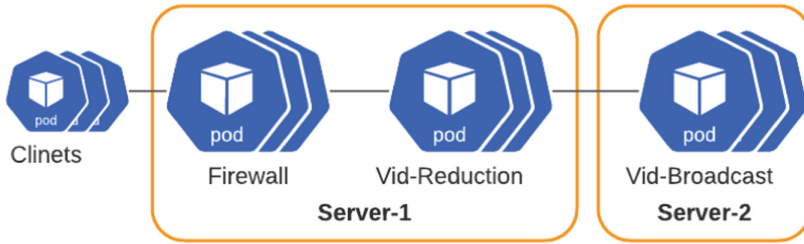


Fig. 5. SFC in Kubernetes multidomain cluster topology

to be associated with, as illustrated in Fig. 4c. The other method is to include a key value when deploying a Pod. The Pod will have an environmental value that is the exact value of the metadata of the network service chain, i.e. ‘SFC-1’, as illustrated in Fig. 4d. After creating the client’s Pod with the correct annotation or environment variable, a request would be sent to the Network Service Manager for connecting the client’s Pod to the specified NS. In return, the Network Service Manager will register the client as an NSC and search in the Network Service Registry for NSE. If an NSE is found in the registry, an interface will be injected into the client and related NSE Pods to create a chain. This chain will include only interconnected Pods, and each Pod will have separate NSM interface(s) where the Pod can communicate with other Pods in the chain.

The chain we developed includes three different service functions. The first service function (Pod) in the chain is the Firewall Pod. This Pod will act as an entry point to the chain in the cluster. Figure 4b illustrates this action. This Pod includes our custom-built container using the Nginx as a base image, and the primary function is to authenticate requests entering the Kubernetes cluster. If the request is allowed to enter the cluster, the traffic will be steered according to the chain identified, as illustrated in Fig. 4b, and the next hop in the chain will be the Vid-Reduction Pod. All traffic that egress the Firewall Pod will traverse to the Vid-Reduction Pod.

The second Pod in the chain is the Vid-Reduction Pod. Traffic from the previous Pod in the chain will ingress into this Pod which is responsible for locating the video file and checking the file size. If the file size is below a threshold, the file will be sent to the next hop in the chain without any modification. Otherwise, the video will be compressed. The Vid-Reduction Pod includes our custom-built container, which was developed using an Nginx base image. Furthermore, we use the FFmpeg tool to perform the compression function for the video file. After the compression function, the compressed file will be sent to the next hop in the chain. The file will leave from the Vid-Reduction Pod and traverse to the next hop in the chain, the Vid-Broadcast Pod.

The third and final Pod in the chain is the Vid-Broadcast Pod. Traffic from the previous Pod, Vid-Reduction Pod, in the chain will ingress into the Pod. This Pod is responsible for broadcasting the requested video to fulfill the client’s request. The Vid-Broadcast Pod container wraps the Nginx RTMP module and

FFMPEG tool. Nginx is open-source software for web servers, reverse proxying, load balancing, streaming, and more [24]. We choose Nginx because it provides a fast and reliable static web server, plus it is one of the most popular web servers.

4 Implementation

Testbed. The experiments were conducted on our testbed, CINE, consisting of 2 servers. One of the servers has a 40-core CPU (Intel Xeon E5-2650 @ 2.30) with one 1GbE NIC (Intel I350), and the other server has a 40-core CPU (Intel Xeon Silver 4114 @ 2.20GHz) with one 1GbE NIC (BRM 5720).

Kubernetes. We focused on using the latest version of Kubernetes. The client version for Kubernetes is 1.21.3, and the server version for Kubernetes is 1.21.1. We deployed Kubernetes on the two servers mentioned above. We also installed kubelet and kubectl. The kubelet is the component that runs on all the machines in the cluster and performs user's requests such as starting a Pod and containers. The latter is the command line to communicate with the cluster. We used the Kubectl tool to create clusters in Kubernetes.

Networking. We choose the NSM to be deployed in our cluster. The new release of NSM (v1.0.0) does not depend on a specific CNI. Therefore, we select Net Weave, a resilient and straightforward network for Kubernetes and its hosted applications [42]. Weave Net creates a virtual network that connects Docker containers across multiple hosts and enables their automatic discovery.

NSM. We tested two versions of the NSM releases. At first, we worked with the v0.2.0 release. This release was only released against Helm version 2. NSM is released through a set of Helm charts, which are easily deployable in the Kubernetes cluster. Release v0.2.0 introduces more features such as interdomain, DNS, security, and improvement to Network Service Endpoint. We also worked on the new release, v1.0.0, which was not officially published when we wrote the paper. Instead, the NSM community is releasing the latest version in phases. We worked with the new release and created a cluster with our custom-built containers to develop a chain of services. Release v1.0.0 added more features and capabilities from the old release, such as supporting different types of payloads (IP and Ethernet), latency reduction, and topology-aware scale.

5 Performance Evaluation

This section will provide a performance evaluation for the three different frameworks under three categories: Operating System (OS)-level virtualization, technology-based aspects, and management flexibility.

There are two different types of OS-level virtualization. The first one is VMs and the second is containers. The difference between VMs and containers is

the level of OS virtualization. Traditional VMs are heavyweight that run guest operating systems with their binaries, libraries, and applications that it services and the VM may be many gigabytes in size. In comparison, containers incur significantly lower overhead than traditional VMs and are gaining increasing attention in recent years [43]. A container shares host OS kernel, binaries and libraries, and they come in megabytes in sizes. Both OVN4NFV-K8s and Contiv-VPP use the VMs technique, which adds management overheads. Developers will have to deal with any additional issues when creating the VMs. While on the other hand, the NSM framework focuses on containers to reduce management overhead because they use the operating system's standard system call interface. But this comes with a flexibility issue where containers are not as flexible as a VM.

Secondly, each framework uses a different technology to present its solution for SFC. The Contiv-VPP extension only supports L2 cross-connect for interconnecting between Pods and only supports one single data path. Contiv-VPP relies on Data Plane Development Kit (DPDK) technology which offloads packet processing from the operating system kernel to userspace. Using DPDK technology brings benefits such as accelerating packet processing workloads. Despite that, it might be challenging to set up the correct environment and install DPDK for Contiv-VPP to function correctly on bare metal servers. On the other side, if you choose to implement Contiv-VPP using VMs, this will eliminate the challenges of installing DPDK as the VMs will be ready to use. Furthermore, Contiv-VPP requires a specific hypervisor, VirtualBox, limiting the users due to lack of support to VirtualBox hypervisor. It is essential to mention that Contiv-VPP only uses memif interfaces. Finally, Contiv-VPP provides a user interface that might help visualize the components and connect them.

OVN4NFV-K8s is based on Open Virtual Network (OVN), which supports virtual network abstraction and complements the existing capabilities of Open vSwitch that provides L2/L3 virtual networking, such as logical switches and routers, multiple tunnel overlays, and L2/L3/L4 ACLs. It is essential to mention that the OVN4NFV-K8s plug-in is a project under the Open Platform for NFV (OPNFV). Hence, it inherits and is limited to the OPNFV features. The third framework, NSM, complements traditional service mesh [44] and provides an infrastructure layer over microservices to standardize the runtime operations of applications. NSM focuses on supporting applications that might consist of many microservices, leading to simplicity, flexibility, and scalability. However, managing different microservices is a complex task, where different languages might be implemented, owned by different tenants, and/or constant changing states to microservices. Finally, comparing NSM with Contiv-VPP and OVN4NFV networking tools, NSM does not alter the Kubernetes CNI; instead, it is a standalone mechanism that consists of several components that can be deployed in a Kubernetes cluster. NSM provides different types of interfaces to be injected in a Pod. It gives the users a choice between using a memif or kernel interface.

The final category in evaluating the performance is each framework's flexibility for developers to configure the framework accordingly. Creating a Kubernetes

cluster using the OVN4NFV-K8s and Contiv-VPP framework was more complex than the NSM framework. Both OVN4NFV-K8s and Contiv-VPP require heavy pre-configuration. For OVN4NFV-K8s, the framework requires specific pre-configuration before deploying the cluster. Plus, it depends on building VMs rather than containers. On the other hand, deploying a Kubernetes cluster using NSM is smooth. We started by using Calico and implemented Calico as CNI with a single node cluster in using NSM. However, Calico causes some issues when switching to two physical node clusters. It delays connections between spire agent and spire server, consequently generating the failure of workload registration in NSM infrastructure. But NSM provides the freedom for users to choose amongst different networking plug-ins. Furthermore, NSM does not require heavy pre-configuration to deploy the networking plug-ins as it provides great flexibility.

6 Limitations and Future Work

We faced multiple challenges during our work to deploy an SFC in Kubernetes. The first challenge was working with NSM releases. Between old and new releases of the NSM framework, the documentation provided is inferior. The latest release of NSM introduces a new method to deploy Network Service. It involves the Kubernetes concepts of the Kustomize tool, a standalone tool to customize Kubernetes objects through a kustomization file [27]. We faced the second challenge of integrating our custom-built containers into the NSM framework. Specifically when adding the feature of injecting memif interfaces to coexists with our service function. SM framework forces the injection of interfaces, and traffic will have to ingress and egress specifically from those interfaces. This can limit service functions types implemented in a chain. The final challenge we faced was the transfer process of the video file between Pods. Transferring files between Pods in a chain is different than regular file transfer using Kubernetes-based features. There are many tools for file transfer, such as the secure copy protocol, but implementing it in a container will increase the container image size. This will eliminate one of the container's benefits of being lightweight.

SFC is still not mature in microservice-based network architecture. More research is needed to provide solutions for chaining service functions while using containers and not traditional VMs. Furthermore, the SFC concept is limited to small applications, such as load balancing and packet investigations. Big science data flow applications might benefit from SFC features if deployed correctly. Another area of improvement is an analytical study of the effect of different network interfaces performance. In our example, we used both kernel and memif interfaces. Ideally, it would be good to analyze how each different interface performs in a container environment. Finally, applying network analysis to extract network metrics and optimize the performance will provide a better QoS over the chain.

7 Related Work

The authors in [28] provide a similar work by using NSM in Kubernetes to offer SFC solutions. They proposed an efficient traffic steering orientation for cloud-native service function chaining. They proposed a new network-aware traffic orientation model based on weighted cycles. This is different from our work as we focus on SFC's design and implementation process using the NSM framework in Kubernetes. Also, in [29], they offered a solution to maximize the QoS satisfaction rate by load-balancing the traffic over the SFC path using the Convtiv-VPP method. Few papers [30,31] focus on integrating OpenStack and Kubernetes to deploy a chain of service functions. OpenStack provides VMs for users to deploy their services and applications, while Kubernetes orchestrates and manages containers. Bringing both OpenStack and Kubernetes together uses Kuryr, an OpenStack project that aims to solve container networking issues in OpenStack. Many papers fill in the gap for container-based orchestration. Since there is no standard for defining container-based VNFs, many articles fill the gap by designing new solutions such as extending Tacker architecture (NFV management and orchestration framework) [32,33]. The authors in [34] proposed a fault management system with dynamic policy recovery enforcement to support the high availability of SFC in a multi-cloud environment. In [35], the authors proposed a performance model approach for recommending an initial resource provisioning for every microservice within all CNFs before deploying the SFC. Another interesting paper [36] proposed a machine learning framework module that can detect anomalies for SFC integrity. Finally, a recent paper [37] proposed a resource and energy-aware SFC strategy in the edge-cloud environment for IoT applications that would cope with dynamic load and resource situations emerging from dynamic SFC requests. Our work is related to those papers mentioned in this section by building a service function chain. We took a different avenue by providing sufficient details on SFC's design and implementation process using the NSM framework in Kubernetes and adding more value to a service function chain. To our knowledge, no previous work used the NSM framework for building a chain of network services using real-life use cases.

8 Conclusion

NFV is the future technology that enables cloud-based platforms to provide public services and acquire resources such as networking, computing, and storage. This concept unfolded innovations such as container-based microservices for deploying services and applications. Containers are efficient and flexible while incurring significantly lower overhead. Kubernetes is a tool to orchestrate and manage containers. Kubernetes' function strategy follows a declarative microservice approach. Kubernetes provides service discovery and load balancing, automation in self-healing, optimal scheduling, and security mechanisms. It also has a shorter time to deployment due to architecture, logging detail and live "in-service" debugging. Kubernetes does provide a specific way to interconnect

Pods and containers. Instead, it depends on the third party to provide overlay network functions such as NSM over Kubernetes essential network functions. These projects follow the Kubernetes networking model to build a networking plug-in for the Kubernetes cluster. We provide details on different networking extensions that support SFC. We also briefly explain the various networking plug-ins that support CNI, such as Calico, Canal, and Contiv-VPP.

We created a Kubernetes cluster using the NSM framework, supporting the SFC concept. We created a service function chain that consisted of multiple Pods in a multi-node cluster. The Pods contained our custom-built containers, and each container was built to perform a different function. The container functions we built were firewall, video compression, and video broadcasting containers. We found limitations when using the NSM framework to deploy the SFC concept. Hence, the limitations of deploying the SFC concept on Kubernetes are related to the functionality and features of the networking extension plug-in we used (NSM). Our SFC design and implementation focused on providing a real-life scenario compared to traditional chains with limited service functions.

Acknowledgement. This project is supported by the Mitacs Accelerate program funded by NSERC between Ciena and Carleton University, Ottawa, Canada.

References

1. Tsuji, Y., Itoh, A., Kobayashi, M.: Future network technologies for the 5G/IoT Era. *NTT Tech. Rev.* **16**(6) (2018)
2. ETSI Industry Specification Group (ISG): Network Functions Virtualisation (NFV): An introduction, benefits, enablers, challenges and call for action. SDN and OpenFlow World Congress, Darmstadt, Germany (2012)
3. ETSI Industry Specification Group (ISG): Network Functions Virtualisation (NFV): Architectural Framework (2014)
4. Halpern, J., Pignataro, C.: Service Function Chaining (SFC) Architecture. In: RFC, number 7665, October 2017–1721, RFC Editor, RFC Editor (2015)
5. Kubernetes, Production-Grade Container Orchestration, <https://kubernetes.io/>. Accessed 15 Nov 2021
6. Cloud Native Computing Foundation, CNCF Survey Report 2020. <https://www.cncf.io/wp-content/uploads/2020/12/CNCF-Survey-Report-2020.pdf>. Accessed 15 Nov 2021
7. Cziva, R., Pezaros, D.P.: Container network functions: bringing NFV to the network edge. *IEEE Commun. Mag.* **55**(6), 24–31 (2017). <https://doi.org/10.1109/MCOM.2017.1601039>
8. Cloud-Native Network Functions. <https://cdnf.io/>. Accessed 15 Nov 2021
9. Li, X., Rao, J., Zhang, H., Callard, A.: Network Slicing with Elastic SFC. In: IEEE 86th Vehicular Technology Conference (VTC-Fall), pp. 1–5 (2017). <https://doi.org/10.1109/VTCFall.2017.8287914>
10. Barakabitze, A.A., et al.: 5G network slicing using SDN and NFV: a survey of taxonomy, architectures and future challenges. *Comput. Netw.* **167**, 106984 (2020). <https://doi.org/10.1016/j.comnet.2019.106984>
11. CNI - the container network interface. <https://github.com/containernetworking/cni>. Accessed 15 Nov 2021

12. Benchmark-k8s-cni-2020-08. <https://github.com/InfraBuilder/benchmark-k8s-cni-2020-08>. Accessed 15 Nov 2021
13. Project Calico. <https://docs.projectcalico.org/getting-started/kubernetes/>. Accessed 15 Nov 2021
14. KOPS -Kubernetes Operation. <https://kops.sigs.k8s.io/networking/canal/>. Accessed 15 Nov 2021
15. Contivpp.<https://contivpp.io/>. Accessed 15 Nov 2021
16. FD.io, The world's secure networking data plane. <https://fd.io/>. Accessed 15 Nov 2021
17. Network Service Mesh. <https://networkservicemesh.io/>. Accessed 15 Nov 2021
18. Custom Resources. <https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/>. Accessed 15 Nov 2021
19. OPNFV/OVN4NFV-K8s-K8s-plugin. <https://github.com/opnfv/ovn4nfv-k8s-plugin/>. Accessed 15 Nov 2021
20. IDG 2020 IDG Cloud Computing Study. <https://resources.idg.com/download/2020-cloud-computing-executive-summary-rl/>. Accessed 15 Nov 2021
21. CDNf, Cloud-Native Network Functions. <https://cdnf.io>. Accessed 15 Nov 2021
22. Docker Homepage. <https://www.docker.com/>. Accessed 15 Nov 2021
23. Container Network Interface specification. <https://github.com/containernetworking/cni/blob/master/SPEC.md>. Accessed 15 Nov 2021
24. NGINX Homepage. <https://www.nginx.com/>. Accessed 15 Nov 2021
25. Kind Homepage. <https://kind.sigs.k8s.io/>. Accessed 15 Nov 2021
26. Simple CNI plugin with IPv4, IPv6 and DualStack support. <https://github.com/aojjea/kindnet>. Accessed 15 Nov 2021
27. Declarative management of Kubernetes objects using kustomize. <https://kubernetes.io/docs/tasks/manage-kubernetes-objects/kustomization/>. Accessed 15 Nov 2021
28. Dab, B., Fajjari, I., Rohon, M., Auboin, C., Diquélou, A.: An efficient traffic steering for cloud-native service function chaining. In: 23rd conference on innovation in clouds, Internet and Networks and Workshops (ICIN), pp. 71–78 (2020). <https://doi.org/10.1109/ICIN48450.2020.9059340>
29. Bouridah, A., Fajjari, I., Aitsaadi, N., Belhadeif, H.: Optimized scalable SFC traffic steering scheme for cloud native based applications. In: IEEE 18th Annual Consumer Communications & Networking Conference (CCNC), pp. 1–6 (2021). <https://doi.org/10.1109/CCNC49032.2021.9369583>
30. Vu, X.T., et al.: An architecture for enabling VNF auto-scaling with flow migration. In: 2020 International Conference on Information and Communication Technology Convergence (ICTC), pp. 624–27. IEEE (2020). <https://doi.org/10.1109/ICTC49870.2020.9289507>
31. Kouchaksaraei, H.R., Karl, H.: Service function chaining across openstack and kubernetes domains. In: Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems (2019)
32. Hoang, C.-P., et al.: An extended virtual network functions manager architecture to support container. In: Proceedings of the 2018 International Conference on Information Science and System, pp. 173–176. ACM (2018). <https://doi.org/10.1145/3209914.3209934>
33. Yang, H., Hoang, C., Kim, Y.: Architecture for virtual network function's high availability in hybrid cloud infrastructure. In: 2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), pp. 1–5 (2018). <https://doi.org/10.1109/NFV-SDN.2018.8725784>

34. Song, S.-Y., Lin, F.J.: Dynamic fault management in service function chaining. In: IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC), pp. 1477–1482. IEEE (2020). <https://doi.org/10.1109/COMPSAC48688.2020.00-46>
35. Khan, M.G., et al.: A performance modelling approach for SLA-aware resource recommendation in cloud native network functions. In: 6th IEEE Conference on Network Softwarization (NetSoft), pp. 292–300 (2020). <https://doi.org/10.1109/NetSoft48620.2020.9165482>
36. Cheng, S.-T., Zhu, C.-Y., Hsu, C.-W., Shih, J.-S.: The anomaly detection mechanism using extreme learning machine for service function chaining. In: 2020 International Computer Symposium (ICS), pp. 310–315 (2020). <https://doi.org/10.1109/ICS51289.2020.00068>
37. Thanh, N.H., Kien, N.T., Van Hoa, N., Huong, T.T., Wamser, F., Hossfeld, T.: Energy-aware service function chain embedding in edge-cloud environments for IoT applications. *IEEE Internet Things J.* **8**(17), 13465–13486 (2021). <https://doi.org/10.1109/JIOT.2021.3064986>
38. Creating a cluster with kubeadm. <https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/>. Accessed 15 Nov 2021
39. Zou, D., Huang, Z., Yuan, B., Chen, H., Jin, H.: Solving anomalies in NFV-SDN based service function chaining composition for IoT network. *IEEE Access* **6**, 62286–62295 (2018). <https://doi.org/10.1109/ACCESS.2018.2876314>
40. Imagane, K., Kanai, K., Katto, J., Tsuda, T., Nakazato, H.: Performance evaluations of multimedia service function chaining in edge clouds. In: 15th IEEE Annual Consumer Communications & Networking Conference (CCNC), pp. 1–4 (2018). <https://doi.org/10.1109/CCNC.2018.8319249>
41. Memif Poll Mode Driver. <https://doc.dpdk.org/guides/nics/memif.html>. Accessed 15 Nov 2021
42. Weaveworks, Integrating Kuberntes via the Addon. <https://www.weave.works/docs/net/latest/kubernetes/kube-addon/>. Accessed 15 Nov 2021
43. Zhang, Q., Liu, L., Pu, C., Dou, Q., Wu, L., Zhou, W.: A comparative study of containers and virtual machines in big data environment. In: IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 178–185 (2018). <https://doi.org/10.1109/CLOUD.2018.00030>
44. Li, W., Lemieux, Y., Gao, J., Zhao, Z., Han, Y.: Service mesh: challenges, state of the art, and future research opportunities. In: 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE), pp. 122–1225 (2019). <https://doi.org/10.1109/SOSE.2019.00026>
45. CONTIV/VPP. <https://github.com/contiv/vpp/tree/master/k8s/examples/sfc>. Accessed 31 Jan 2022

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

